Architectural Considerations for Highly Scalable Computing to Support On-demand Video Analytics

George Mathew
Lincoln Laboratory
Massachusetts Institute of Technology
Lexington, MA 02420, USA
George.Mathew@ll.mit.edu

***Abstract*—The processing demands on video analytics calls for special design considerations to achieve scalability. Numerous factors influence the running time of an analytic job. The time consumed for raw computing can be improved by well-engineered approaches to execute certain sub-tasks. High scalability can be achieved by selectively distributing computational components. We elucidate such factors that aid scalability and present design choices for architecting them. The principles outlined in this research were used to implement a distributed on-demand video analytics system that was prototyped for the use of forensics investigators in law enforcement. The system was tested in the wild using video files as well as a commercial Video Management System supporting more than 100 surveillance cameras as video sources. The architectural considerations of this system are presented. Issues to be reckoned with in implementing a scalable distributed on-demand video analytics system are highlighted. The bottlenecks and possible solutions are also touched upon.**

Keywords—video analytics; on-demand video intelligence; intelligent video system; video analytics platform

I. INTRODUCTION

Video Analytics systems has been of tremendous interest in various fields due to its utility [1]. A survey of Intelligent Video Systems and Analytics [2] highlights the importance of such systems in a variety of domains. The richness of information in video data is invaluable for surveillance and forensics. Without the help of a video analytics system, a forensics investigator has to manually sift through large amounts of video clips, which is a tedious and error prone activity [3]. Video Analytics systems fall into two general categories: real-time and on-demand. In real-time systems, the analytics is run continuously on the video data feed. For example, edge cameras can be designed to run real-time analytics [4]. These systems are typically designed for real-time surveillance [5, 6]. However, many investigative analytics are performed on video clips in (recent) history. On-demand video analytics systems provide the capability to execute analytics on an as and when needed basis on any video source for a specific day/time period from the past. The focus of this work is on-demand video analytics systems. There are two sources of videos available to an on-demand system. One is video files and the other is Video Management Systems (VMS's). Video files are typically stored in mp4 or similar formats on disks and directly accessible for video analytics. Citizens contributed videos account for a good number of law enforcement investigative artifacts and hence the significance of video files. VMS's are mostly commercial systems that manage 100's of edge cameras. For the use of an analytic, the video clip for a specific time period has to be accessed programmatically through the VMS using an SDK provided by the commercial vendor.

The Video Analytics System targeted in this work is meant to support multiple users, multiple video sources and multiple analytics. A Video Analytics System can be centralized or distributed in nature. A centralized system can perform only within the limits of its computing resources. Vertical scaling hits the ceiling within the computing power of a single server. To go beyond that, horizontal scaling is necessary and hence we embrace a distributed architecture. To emphasize the scale of the system under consideration, in the rest of the paper, we will use the term Video Analytics Platform (VAP) instead of Video Analytics System.

The essential components of the on-demand VAP is shown in Fig. 1. The system does not run analytics on the edge cameras; instead, all analytics are run inside the platform itself.

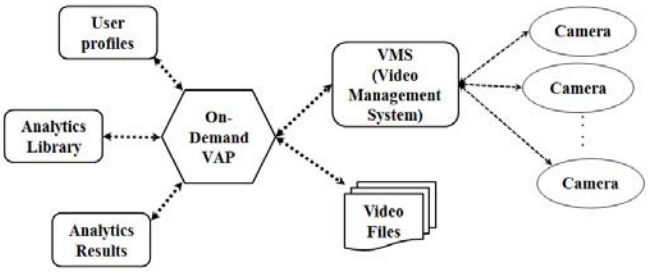

Fig. 1. Components of the On-demand Video Analytics Platform

The working of the VAP in a nutshell is as follows: A user logs in, chooses the video sources of interest and runs analytics on specific time spans on those video sources. The results are stored in the VAP for review and annotations. The user can organize the results and review them as necessary.

II. SCALABILITY FACTORS

User access controls in VMS systems have to be respected by the VAP. Within a VMS, each user is granted permissions to a set of cameras. For obvious security reasons, the VAP

should not by-pass this access control. Consequently, users should have single sign-on into VAP using the VMS's as identity providers. The single sign-on is also significant due to the fact that the VMS SDK's require a logged in user 'session' to fetch video frames for the analytic. If multiple analytics are run concurrently, multiple connection sessions have to be maintained by the VAP. Consequently, for a highly scalable VAP, a 'session broker' is needed for the management of connection sessions. In our implementation, the 'session broker' was implemented as a Video Service that ran on a dedicated server for scalability. A related scalability factor is concurrent video frames pulled from VMS. Table I shows the average time for seeking through video frames in 500 milliseconds intervals on cameras with varying resolutions from a VMS.

TABLE I. AVERAGE SEEK TIMES OVER 500MILS INTERVAL

Camera Resolution	Avg. seek time for a frame over concurrent processes			
	1 process	*2 processes*	*3 processes*	*5 processes*
1920x1440	2.32 secs	2.36 secs	2.41 secs	2.46 secs
1280x960	0.74 secs	0.77 secs	0.79 secs	0.79 secs
480x360	0.22 secs	0.22 secs	0.22 secs	0.23 secs

When multiple analytics are run concurrently, each analytic will need to seek to video frames of interest. Table I shows the worst case scenario in continuously seeking through frames concurrently via independent processes. In reality, an analytic will seek a frame, process it and take necessary action before the next frame is fetched. For scalability, there are some options. If the VMS SDK is thread-safe, it is possible to run a specific analytic pass in its own thread. If not, running each analytic pass in a separate process is an option. As can be surmised from values in Table I, this option has slight performance penalty. Another option for non-thread-safe SDK is to cache ahead the next frame sought while the current frame is being analyzed.

The video sources should have permanent unique identifiers in a VAP. The top-down hierarchy for video sources in the VAP is: VMS → Recorder → Cameras. For the VAP, we tagged the VMS level a 'site'. The recorder level is similar to a 'group'. Each recorder assigns a unique id for a camera (which is not necessarily unique across recorders). Capturing the essence of this format into a label for each camera is necessary for scalability. Also, to maintain consistency with this structure, the File collections have to be organized into 'sites' for various file servers, 'groups' for folders (or directories) and a uniquely generated id as 'camera id' for each file. This gives a unique string representation for each file/camera in the format <site>:<group>:<cameraId>. In our VAP implementation, this string format for files were generated using 'site' as an incrementing positive integer, 'group' and 'camera id' as GUID's [7].

VMS's maintain metadata about the edge cameras. To maintain consistency, video files loaded into the VAP should capture a minimal set of metadata for each file. In our implementation, the following minimal set of information was captured for each video file: display name, file format, video codec, location of the file, start time of the video and end time of the video. From a software developers perspective, to accommodate multiple VMS's and Video Files in a uniform manner, a layer of abstraction is needed in the VAP so that analytics can make the same API call for fetching frames independent of the underlying vendor system. In our implementation, the interface was called 'framegrabber' and it was implemented using ffmpeg [8] for video files and VMS SDK calls for VMS. Direct Show Filter supported by most VMS vendors is another option.

III. FACTORS INFLUENCING ANALYTICS PROCESSING TIME

With advances in high resolution cameras comes the side effects of bigger video frames and higher processing delta. As can be gathered from Table I, bigger frames of higher resolution cameras result in more latency to retrieve frames. If downsampling [9] is supported on the VMS side, it is possible to request frames at a lower resolution for faster frame retrieval. One of the analytics we use in the VAP is video-summarization [10]. This tool creates video summary views with time compression rates up to the 1000's. For practical time calculations, it can be considered a single pass analytic; i.e., it can process the frames for the duration of the video clip in succession from the beginning in one pass. Using video-summarization, the time to capture frames from the VMS and time for processing the frames in a single thread as relative percentages across 3 different cameras (with resolutions as in Table I) are shown in Fig. 2.

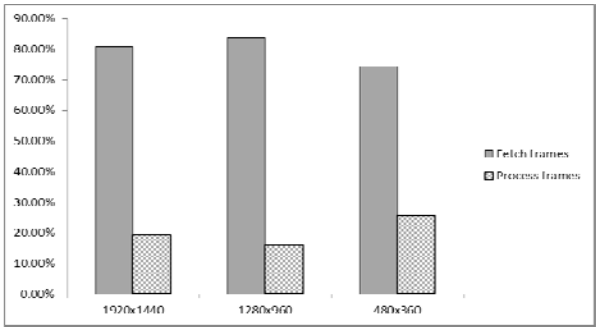

Fig. 2. Split up of time for an anlytic run

It is observed from this graph that irrespective of the camera resolution, approximately 75-85% of the analytics time is spent in capturing video frames. Given the fact that concurrent fetching of frames add only little penalty (from Table I), a possible solution to reduce run time of an analytic is to split the video into multiple clips and run the analytic on each clip and converge the results. This requires an 'orchestrator' component to manage the split and merge. Note that this may not be an option for certain analytic. In those cases where the split-n-mere strategy will work, the individual split video clips have to be processed concurrently. To improve overall processing time, it will be advantageous to run each slice processing on a computing node. Obviously when multiple concurrent jobs are scheduled on same node, the turnaround time for each job increases. This is another reason for having multiple computing nodes. A video analytic job is comprised of the algorithm processing part and the results processing part (for rendering results). In our implementation

project, we used the term ‘compute node’ for an algorithm processing node and ‘transform node’ for a results processing node. For scalability, the architectural design for instrumenting compute/transform nodes have to take into consideration various factors, which are addressed in the next section.

IV. Compute/Transform Nodes

In order to run algorithm processing on compute nodes and results processing on transform nodes, an analytic job has to be split into two tasks – compute task and transform task. An ‘orchestrator’ service is required to handle this. The Orchestrator assigns job-id’s and task-id’s. Also, packaging the parameters for the tasks is handled by Orchestrator. The parameters for an analytics task have to be passed to a compute/transform node in a node-independent way. In our implementation, we used JSON [11] for encoding and passing parameters. Since JSON is a cross-platform data exchange format, this has the added benefit that parameters for analytics can be passed on to multiple operating systems of choice (windows or Linux) in a uniform fashion. This helps in running analytics on multiple OS’s. A sample JSON payload for a compute task from our implementation is given below:

```
{"Job-Id":"c620b2ac-a208-415c-99a1-cb978a0fa3bc",
    "Task": {
       "Id": "2c7fa763ce2a44448130e82662a15fea",
       "Analytics": "Summarization",
       "Arguments": {
        "VideoSources": [
           {"Id":"4d13763d1607:3e1a7a2e:141",
            "DisplayName": "Bldg. C6 - West"
           }],
        "BeginTime": "2016-10-26T10:00:00",
        "EndTime": "2016-10-26T10:30:00",
        "UserProfile": {
            "Name": “user1",
            "VAPHandle": "d4f5d22d-70f0-4381-9830-cf4d5"
} } } }
```

The Orchestrator has to pass these parameter constructs to the compute/transform run time. In order to make that happen, a small footprint agent runs as a proxy for the compute/transform runtime and when the parameters are received, the agent bootstraps the runtime. There are 2 options for this. One is a pull model and the other is a push model. In the pull model, the parties poll for information. In the push model, the parties are handed over the requisite information.

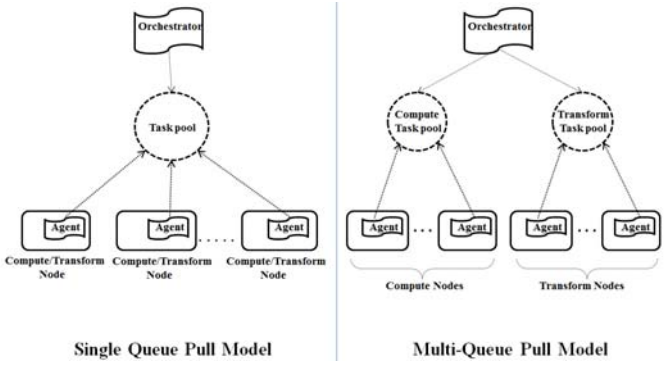

Fig. 3. Pull models for task distribution

As depicted in Fig. 3., the Orchestrator puts the task parameters (in JSON format) in a Task pool. The agents poll the Task pool periodically to check for new tasks coming into the pool. A new task is executed by the first agent who picks it from the pool. The pull model can be implemented in two ways. In the first method, there is only one task pool shared between compute tasks and transform tasks. Any task can be picked up by an agent. The task can be a compute task or a transform task. Every node can execute a compute task or a transform task. In the other model, there are two pools of tasks: one for compute tasks & the other for transform tasks. The agent can listen on either one of the task pools. If it listens on the compute (transform) task pool, it makes that node a compute (transform) node. Essentially, this self-subscription splits the nodes into two groups of compute and transform nodes. The Orchestrator has to be aware of the types of pools and put the task in the right pool. We tried both models and chose the multi-queue pull model by dedicating a certain number of compute nodes and a fixed number of transform nodes. This was primarily because the transform nodes didn’t need as much resources as the compute nodes. In our implementation, we used RabbitMQ for the task pool; with one message queue for compute tasks and another message queue for transform tasks. The pull model is very flexible and scalable. Any new node can be added to listen on the task pool and any idle node can be removed with no loss of operation.

In the push model (see Fig. 4), a master task scheduler assigns each task to an agent. The master task scheduler maintains a resource pool of compute nodes and transform nodes. The Orchestrator hands over tasks to the master task scheduler.

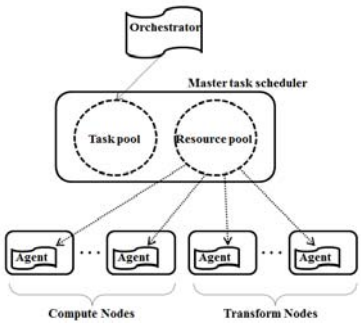

Fig. 4. Push model for task distribution

The Orchestrator has to instruct the master task scheduler which resource pool to use for a given task. The master task scheduler maintains resource pools of nodes that are registered as compute nodes or transform nodes. In this model, a new node has to be registered with a resource pool. Removing a node from the resource pool also means de-listing with the master task scheduler. Note that it is also possible to use a single resource pool and have all nodes run either compute or transform tasks similar to the single-queue pull model. We used htcondor [12] job scheduler for evaluation of the push model.

Both pull and push models provide computing scalability. However, the pull model is quite simple. The push model is based on job schedulers from High Performance Computing

domain. These job schedulers have rich set of task management capabilities. Due to the simplicity of the pull model, in our VAP deployment in the wild, we used the pull model. In our pull model, we also provided a mechanism for a failed task to be put back into the Task pool for a second retry. If the task fails on second attempt also, the job is discarded.

In order for the compute nodes and transform nodes to exchange data seamlessly, it is necessary to have shared storage. A simple implementation strategy is to use NAS (Network Attached Storage) and map a common share between windows and Linux nodes. Another implementation detail is that the algorithm processing part of an analytics must run to completion and save the results even if the user logs out of the VAP or user session times out.

V. Implementation Specifics

The implementation of our VAP was called SIGMA (Scalable Integration of Geo-tagged Monitoring Assets). The various components of the SIGMA environment are shown in Fig. 5.

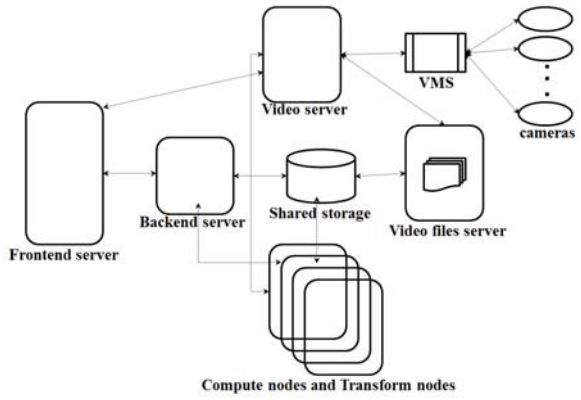

Fig. 5. Architecture of the implemented SIGMA platform

The frontend server hosted a web server (nginx) and the SSO server (OpenAM). The OpenAM layer was used as a gateway between SIGMA and the VMS for user authentication. The backend server hosted the amqp message queue service (RabbitMQ) and mongoDB database. A video server maintained the session broker for user connections to VMS's. The Video server also had the interface for fetching frames from video sources. Metadata for video files as well as analytics results were stored in mongoDB. The compute nodes and transform nodes had agents with small footprint running on them polling the RabbitMQ for tasks to be executed. Two queues were maintained on RabbitMQ – one for compute agents and one for transform agents. Each user could create workspaces to store the analytic results. Workspaces could be shared between users. The shared storage was made available to windows nodes as \\sigmafs\data CIFS share and to Linux nodes as /sigmafs/data NFS mount point.

VI. Conclusion

The scalability of an on-demand video analytics platform that supports multiple users, multiple analytics and multiple video sources depends on various factors that influence different components of the VAP. We discussed those factors and the corresponding architectural design choices. The implementation details of an on-demand Video Analytics Platform implemented using these principles was outlined. This platform was distributed in nature and comprised of multiple service nodes including compute nodes and transform nodes that handled different processing needs of video analytics. The utility of the platform was demonstrated by its deployment in the wild for law enforcement investigators using video files and commercial VMS with more than 100 video cameras attached. We also delved into mechanisms to improve analytics turnaround. Areas of possible bottlenecks and methods to overcome them were also discussed.

Acknowledgment

Distribution A: public release; unlimited distribution. This material is based upon work supported by the Department of Homeland Security Science and Technology Directorate under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Homeland Security.

References

[1] H. Liu, S. Chen, and N. Kubota, "Guest editorial special section on intelligent video systems and analytics", *IEEE Ttrans. Ind. Inf.*, vol. 8, no. 1, p. 90, Feb. 2012.

[2] L. Honghai, C. Shengyong, and K. Naoyuki, "Intelligent Video Systems and Analytics: A Survey", IEEE Trans. Ind. Inf., vol. 9, pp. 1222-1233, Aug. 2013.

[3] H. Arun, B. Lisa, C. Jonathan, E. Ahmet, H. Norman, L. Max, M. Hans, P. Sharath, S. Andrew, S. Chiao-Fe, and T. Ying, "Smart Video Surveillance", IEEE Signal Processing Magazine, pp. 38-51, Mar. 2005.

[4] S. Zhang, S. C. Chan, R. D. Qiu, K. T. Ng, Y. S. Hung, and W. Lu, "Om the Design and Implementation of a High Definition Multi-view Intelligent Video Surveillance System", IEEE International Conference on Signal Processing, Communication and Computing, pp. 353-357, Aug. 2012.

[5] S. Muller-Schneiders, T. Jager, H. S. Loos, and W. Niem, "Performance Evaluation of a Real Time Video Surveillance", 2nd Joint IEEE International Workshop on Visual Surveillance and Performance Evaulation of Tracking and Surveillance, pp. 137-144, Oct. 2005.

[6] M. Bramberger, J. Brunner, B. Rinner, and H. Schwabach, "Real-time Video Analysis on an Embedded Smart Camera for Traffic Surveillance", 10th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 174-181, May. 2004.

[7] P. Leach, M. Mealling, and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace", https://www.ietf.org/rfc/rfc4122.txt, Jul. 2005.

[8] FFmpeg – home page, https://www.ffmpeg.org, accessed Nov. 2016.

[9] V. Nguyen, Y. Tan, and W. Lin, "Adaptive downsampling/upsampling for better video compression at low bit rate", IEEE International Symposium on Circuits and Systems, pp. 1624-1627, May. 2008.

[10] N. Gallo, and J. Thorton, "Fast Dynamic Video Content Exploration", IEEE International Conference on Technologies for Homeland Security, pp. 271-277, Nov. 2013.

[11] E. T. Bray, "The Javascript Object Notation (JSON) Data Interchange Format", https://www.ietf.org/rfc/rfc7159.txt, Mar. 2014.

[12] HTCondor – Home page, https://research.cs.wisc.edu/htcondor/", accessed Nov. 2016.